Angular life cycle hooks:

Angular Component Lifecycle Hooks: An Overview

Hello there! Have you ever wondered how Angular handles the magic that happens behind the scenes in your web apps? Let us now dive into an insightful [Angular tutorial](https://www.dotnettricks.com/learn/angular) and discuss Angular Component Lifecycle Hooks! These amazing occurrences occur at various phases, causing your components to do wonderful things. It's like getting a backstage pass to see what's going on in your app, all made clearer through comprehensive [Angular certification training](https://www.scholarhat.com/training/angular-certification-training)!

What is Angular Component Lifecycle Hooks?

Lifecycle hooks are Angular methods that are executed at certain points during a component's lifecycle. These methods allow you to tap into the Angular component lifecycle and apply custom logic or operations at specified points in time.

Eight Lifecycle Hooks in Angular

Angular has many lifecycle hooks that allow you to inject code at precise times in a component's or directive's life cycle. These hooks function similarly to checkpoints, allowing you to conduct specified tasks based on the current state of the component. In Angular, there are eight lifecycle hooks:

* 1. ngOnChanges
  2. ngOnInit
  3. ngDoCheck
  4. ngAfterContentInit
  5. ngAfterContentChecked
  6. ngAfterViewInit
  7. ngAfterViewChecked
  8. ngOnDestroy

1. ngOnChanges

When the value of an input binding to the component changes, this hook is called.

*Example of ngOnChanges*

**ngOnChanges**(changes: SimpleChanges) {  
**if** (changes['data']) {  
this.**processData**(changes['data'].currentValue);  
}  
}This example determines whether or not the data input binding has changed. If it has, it uses the new data value to run the processData method.

2. ngOnInit

Once the component has been initialized and its input bindings have been handled, this hook is invoked.

*Example of ngOnInit*

**ngOnInit**() {  
this.initialData = this.**getData**();  
}After the component has been properly initialized, this example retrieves initial data from the getData function and stores it in the initialData property.

3. ngDoCheck

Every change detection cycle ends with a call to this hook. Because of the performance ramifications, it is frequently regarded as an anti-pattern.

*Example of ngDoCheck*

ngDoCheck() { **if** (**this**.element.clientWidth !== **this**.lastWidth) { **this**.adjustLayout(); **this**.lastWidth = **this**.element.clientWidth; }}This example checks to see if the width of the element has changed. If so, the layout is adjusted and the lastWidth attribute is updated. This is not advised due to potential performance difficulties.

4. ngAfterContentInit

After the projected content (content projected into the component with <ng-content>) has been initialized, this hook is invoked.

*Example of ngAfterContentInit*

**ngAfterContentInit**() { this.content = this.contentElement.nativeElement.textContent;}This example uses contentElement to retrieve the content projected into the component and store it in its content property.

5. ngAfterContentChecked

This hook is invoked at the end of each change detection cycle for the projected content.

*Example of ngAfterContentChecked*

ngAfterContentChecked() { **if** (**this**.content) { **this**.analyzeContent(**this**.content); }}This example checks whether or not planned content exists as well as if so, analyses it.

6. ngAfterViewInit

After the component's view (including its children) has been fully initialized, this hook is called.

*Example of ngAfterViewInit*

ngAfterViewInit() { **this**.canvas = **this**.canvasElement.nativeElement; **this**.drawChart(**this**.**data**);}This example retrieves the canvas element & utilizes it to create a chart using the data provided.

7. ngAfterViewChecked

This hook is invoked at the end of each change detection cycle for the component's view.

*Example of ngAfterViewChecked*

**ngAfterViewChecked**() { **if** (this.isScrolling) { this.**updateScrollPosition**(); }}This example determines whether the component is actively scrolling and, if so, adjusts the scroll position.

8. ngOnDestroy

When the component is destroyed, this hook is called.

*Example of ngOnDestroy*

**ngOnDestroy**() { this.**unsubscribeFromEvents**();}To prevent memory leaks, this example unsubscribes from any event subscriptions.

Summary

Component creation, update, & destruction all involve angular magic. Lifecycle hooks enable checkpoints for injecting code at certain stages in the app's lifecycle, improving app functionality and speed. Make good use of these hooks for a smooth and efficient Angular experience!

Angular provides component based architecture that allows modularizing the application. It means you can create multiple chunks, and convert your large component to a smaller segment that can be easily maintained. The main advantage to do this is easily understanding the component and maintenance of code. In this scenario, it is important for communication between components or share the data between the component.

Angular provides various ways to share the data between component

* + Using @Input decorator (Parent to Child)
  + Using @Output decorator and EventEmitter (Child to Parent)
  + Using @ViewChild decorator (Child to Parent)
  + Using a Service (Unrelated Components)

Using @Input decorator (Parent to Child)

When we define a variable with the @Input decorator in the component, it allows that variable value can get from the parent component template. The definition of the @Input decorator is defined in '@angular/core' module.

In the following example, the child component contains the variable "componentTitle" that declare with @Input decorator and this variable can be set from the parent component.

**child-component.component.ts**

import { Component, Input, OnInit} from '@angular/core';  
  
@Component({  
selector: 'app-child-component',  
templateUrl: './child-component.component.html',  
styleUrls: ['./child-component.component.css']  
})  
export class ChildComponentComponent implements OnInit {  
@Input() componentTitle: string;  
constructor() { }  
ngOnInit() {  
}  
}

JavaScript

Copy

**child-component.component.html**

<p>  
child-component:  
</p>  
<p>  
Data: {{componentTitle}}  
</p>

Markup

Copy

**parent-component.component.html**

<p>  
Parent Component  
</p>  
<br/>  
<input [(ngModel)]="title" class="form-control Report-1" type="text">  
<app-child-component componentTitle="{{title}}" ></app-child-component>

Markup

Copy

**parent-component.component.ts**

import { Component, OnInit } from '@angular/core';  
@Component({  
selector: 'app-parent-component',  
templateUrl: './parent-component.component.html',  
styleUrls: ['./parent-component.component.css']  
})  
export class ParentComponentComponent implements OnInit {  
title: string;  
constructor() { }  
ngOnInit() {  
}  
}

JavaScript

Copy

Using @Output decorator and EventEmitter (Child to Parent)

To pass the data from the child component, we have to emit the data from child component and parent component will be listening for the event to get the data from Child component.

In the following example, the child component has "emitData" emitter that send the data to the parent component when user click on the "Send the Data to Parent" button. To receive the data to parent component, we have to create method to grab the data. The parent component listens for the event and when message is emitted by the child component, it captures the data in $event.

**child-component.component.ts**

import { Component, Input, OnInit, Output, EventEmitter} from '@angular/core';  
  
@Component({  
selector: 'app-child-component',  
templateUrl: './child-component.component.html',  
styleUrls: ['./child-component.component.css']  
})  
export class ChildComponentComponent implements OnInit {  
@Output() emitData = new EventEmitter<string> ();  
message:string = '11';  
constructor() { }  
ngOnInit() {  
}  
getMessage() {  
return this.message;  
}  
sendData() {  
this.message = "Button Click !!!";  
this.emitData.emit("Button Click!!!");  
}  
}

JavaScript

Copy

**child-component.component.html**

<p>  
child-component:  
</p>  
<br/>  
<button (click)="sendData()">Send the Data to Parent </button>

Markup

Copy

**parent-component.component.html**

<p>  
Parent Component  
</p>  
<br/>  
<input [(ngModel)]="title" class="form-control Report-1" type="text">  
<app-child-component (emitData)="emittedDataByChild($event)"></app-child-component>

Markup

Copy

**parent-component.component.ts**

import { Component, OnInit } from '@angular/core';  
  
@Component({  
selector: 'app-parent-component',  
templateUrl: './parent-component.component.html',  
styleUrls: ['./parent-component.component.css']  
})  
export class ParentComponentComponent implements OnInit {  
title: string;  
childData: string;  
constructor() { }  
ngOnInit() {  
}  
emittedDataByChild(data) {  
this.childData = data;  
}  
}

JavaScript

Copy

Using @ViewChild decorator (Child to Parent)

Another simple way to send the data is to use @ViewChild decorator. The @ViewChild decorator help to access a directive, child component and the DOM element from a component class. The @ViewChild decorator returns the element that match reference selector for defined directive, template or component. Using @ViewChild decorator, you can call a child component's method any time after the child component has got loaded. You can access the child component 's properties or method or after the "AfterViewInit" method of the parent component.

In the following example, child component instance has been created in parent component and when user click on "Get Message From Child" button, it calls "getMessage" method of child component.

**parent-component.component.ts**

import { Component, OnInit, ViewChild, AfterViewInit } from '@angular/core';  
import { ChildComponentComponent } from '../child-component/child-component.component';  
  
@Component({  
selector: 'app-parent-component1',  
templateUrl: './parent-component1.component.html',  
styleUrls: ['./parent-component1.component.css']  
})  
export class ParentComponent1Component implements OnInit, AfterViewInit {  
@ViewChild('child') private child: ChildComponentComponent;  
message: string = '';  
newName: string;  
constructor() { }  
ngOnInit() {  
this.dataservice.name.subscribe(data=>{  
this.newName = data;  
});  
}  
ngAfterViewInit() {  
}  
getChildMessage() {  
this.message = this.child.getMessage();  
}  
}

JavaScript

Copy

**parent-component.component.html**

<p>  
parent-component1!  
</p>  
<app-child-component #child></app-child-component>

Markup

Copy

Using Services

All above described methods can be used to share the data between parent to child and vice versa. It becomes a little tricky when sharing the data between unrelated component or sibling component.

Using BehaviorSubject, you can be able to send the data to the sibling component. It holds the data that needs to be shared with other. The component which wants to get change notification, they have to subscribe to the BehaviorSubject instance. You can publish your data change using "next" method. When next method is called, all the subscribers get notified with new data.

In the following example, value change publishes by component1, component2 gets the change notification as it has subscribed to it.

**data.Service.ts**

import { Injectable } from '@angular/core';  
import { BehaviorSubject } from 'rxjs/BehaviorSubject';  
@Injectable()  
export class DataService {  
private nameSource = new BehaviorSubject<string>('');  
name = this.nameSource.asObservable()  
constructor() { }  
changeName(name: string) {  
this.nameSource.next(name);  
}  
}

JavaScript

Copy

**parent-component.component.ts**

import { Component, OnInit } from '@angular/core';  
import { DataService } from '../Services/data.Service';  
  
@Component({  
selector: 'app-parent-component',  
templateUrl: './parent-component.component.html',  
styleUrls: ['./parent-component.component.css']  
})  
export class ParentComponentComponent implements OnInit {  
name: string;  
constructor(private dataservice: DataService) { }  
ngOnInit() {  
}  
changeName() {  
this.dataservice.changeName(this.name);  
}  
}

JavaScript

Copy

**parent-component1.component.ts**

import { Component, OnInit } from '@angular/core';  
import { ChildComponentComponent } from '../child-component/child-component.component';  
import { DataService } from '../Services/data.Service';  
@Component({  
selector: 'app-parent-component1',  
templateUrl: './parent-component1.component.html',  
styleUrls: ['./parent-component1.component.css']  
})  
export class ParentComponent1Component implements OnInit {  
newName: string;  
constructor(private dataservice: DataService) { }  
ngOnInit() {  
this.dataservice.name.subscribe(data=>{  
this.newName = data;  
});  
}  
}

Promise vs Observable

Promise

A [Promise](https://www.geeksforgeeks.org/javascript-promise/) represents a single value in the future, that may not be available at present but is expected to be resolved or rejected in the future. It is more readable and maintainable in asynchronous. A Promise object has two possible states, i.e. the **resolve** and **reject**. It offers a structured way to handle resolved or rejected states. It has “[**then ()**](https://www.geeksforgeeks.org/javascript-promise-then-method/)**” to handle resolved states and “**[**catch ()**](https://www.geeksforgeeks.org/javascript-promise-catch-method/)**” to handle rejected ones. These help in making promises a suitable choice for single asynchronous operations. Suitable for activities such as reading data from server, files. There are 4 phases in it, namely, fulfilled, rejected, pending, and settled.**

* + **Pending**: In this, action is still pending and not yet fulfilled or rejected.
  + **Fulfilled**: This state represents that the asynchronous operation is successfully completed.
  + **Rejected**: In this action is rejected or failed.
  + **Settled**: In this result is determined successfully, either fulfilled or rejected.

Disadvantages of Promises

* + These promises are not cancellable in between the process. Once implemented, we must wait till a result is obtained. i.e., settled.
  + These execute only once and don’t repeat them again.
  + Multiple values are not retrieved over time.
  + When working on large applications, it’s complicated.

Observable

An [Observable](https://www.geeksforgeeks.org/angular-7-observables/) is a powerful concept that is in Angular through the [*RxJS library*](https://www.geeksforgeeks.org/rxjs-beginners-guide/)*. These are a sequence of values that can be arrived at over time. These can emit values either synchronously or asynchronously. Mainly used to handle streams of data that change over time. These can emit multiple values, unlike promises. These offer a great set of complex features such as cancellation, retrying, and debouncing. Suitable for real-time updates like stock market dashboards.*

Disadvantages of Observables

* + These are harder to learn than promises as it’s from RxJS.
  + Debugging is way harder when using Observables.
  + Most times, we don’t require what all observables offer.

Difference between Promise and Observable

|  |  |  |
| --- | --- | --- |
|  | **Promise** | **Observable** |
| **Handling multiple values** | Handles single value. | Handle multiple values at a time. |
| **Asynchronous support** | Suitable for asynchronous communication | Suitable for both synchronous and asynchronous communication |
| **Cancellation** | Cannot be canceled once initiated. | Can be canceled whenever we want. |
| **Complex data transformation** | Limited support. | Wide range of support. |
| **Error Handling** | The catch() method is used for handling errors. | This offers different mechanisms. |
| **Conciseness** | Simple and concise syntax. | More complex due to extensive support. |
| **Use Cases** | Suitable for one-time tasks like reading files. | Suitable for continuous real-time updates like in stock market dashboards. |

**Deploying an Angular App on IIS and Azure**

Certainly! Let’s explore how to deploy Angular applications on both **Azure** and **IIS**.

Deploying an Angular App on Azure Static Web Apps

Azure Static Web Apps is a great platform for hosting static websites, including Angular applications. Here are the steps to deploy an Angular app using Azure Static Web Apps:

1. **Prerequisites**:
   * **Azure subscription**: If you don’t have one, create a free trial account.
   * **GitHub account**: You’ll need this to link your repository.
   1. **Create a Repository**:
      * Use a GitHub template repository to get started. For example, you can use the [Angular Basic template](https://github.com/staticwebdev/angular-basic/generate).
      * Name your repository (e.g., my-first-static-web-app).

**Create a Static Web App**:

Go to the Azure portal.

Search for “Static Web App” and select it.

Configure your app by linking it to your GitHub repository.

Specify the resource group, name, plan type, and region.

Authenticate with GitHub and provide the repository details (organization, repository name, and branch).

**Deploy the App**:

Azure Static Web Apps will automatically build and deploy your Angular app.

Once deployed, you’ll get a URL for your app.

For detailed instructions, refer to the [official Microsoft Learn guide on deploying Angular apps to Azure Static Web Apps1](https://learn.microsoft.com/en-us/azure/static-web-apps/deploy-angular).

**Deploying an Angular App on IIS**

If you prefer to host your Angular app on an IIS server, follow these steps:

**Install IIS**:

Make sure you have Internet Information Services (IIS) installed on your Windows server.

**Build Your Angular App**:

Build your Angular app using the ng build command.

This generates the production-ready files in the dist folder.

**Deploy to IIS**:

Copy the contents of the dist folder to the web root directory of your IIS server.

You can use FTP or any other method to transfer the files.

Ensure that your IIS site points to the correct folder.

**Configure URL Rewrite** (Optional):

If your Angular app uses client-side routing (e.g., Angular Router), you’ll need to configure URL rewriting.

Install the URL Rewrite Module for IIS.

Create a web.config file in your app’s root directory with the necessary rewrite rules.

That’s it! Your Angular app should now be accessible via IIS.

For a more detailed walkthrough, check out this [Medium article](https://medium.com/angular-in-depth/deploy-an-angular-application-to-iis-60a0897742e7) [on deploying Angular apps to IIS](https://learn.microsoft.com/en-us/azure/static-web-apps/deploy-angular)[2](https://medium.com/angular-in-depth/deploy-an-angular-application-to-iis-60a0897742e7).

Remember to choose the deployment method that best suits your needs and infrastructure. Azure Static Web Apps is great for serverless hosting, while IIS provides more control over your server environment. Happy deploying! 😊🚀

**Interceptors**

What is Interceptor in Angular? 
What is the use of Interceptors in 
Angular? 
Creating HTTP Interceptors in Angula 
1. Add Request Header using 
Interceptors 
2. Cross-Site Request Forgery (XSRF 
or CSRF) protection by Interceptors 
3. Global Error Handling using 
Interceptors 
4. Using Angular Interceptors for 
Logging or Profiling 
5. Mock Backend Using Angular 
Interceptors 
6. Change Url by Interceptors 
Summary 

Angular Interceptors are another important component of application development. They are used to alter HTTP requests by including various functions.

Interceptors are an excellent way to modify the HTTP request or response to make it more simple and understandable.

What Is An Angular HTTP Interceptor

An HTTP interceptor is an Angular service that intercepts HTTP requests and responses generated by the built-in HTTP client of the Angular framework.

By using an interceptor to change HTTP requests and answers in a single area, we may avoid redundant code and make our code more intelligible.

Angular Interceptor is built similarly to other services, but it must have an intercept function. You will always intercept the request and, if desired, follow it through to intercept the response.

How Does The Interceptor Work?

Assume we need to do some operation on each request sent through httpClient in the application.

For example, we must provide an auth token in each request so that the server can authenticate and deliver us the required answer. To do this, we will establish a service in between that will allow us to modify our httpRequest or httpResponse accordingly. The graphic below might help you better grasp the situation:

How To Create A Simple HTTP Interceptor

Angular interceptor can be created using the following command:

ng generate interceptor ExampleInterceptor

An angular interceptor will look the same as a service with @Injectable(), which implements HttpInterceptor, which has a method called intercept that allows you to intercept the HTTP request and response. Consider the following example:

import { Injectable } from '@angular/core';

import { HttpInterceptor, HttpEvent, HttpRequest, HttpHandler } from '@angular/common/http';  
import { Observable } from 'rxjs';  
  
@Injectable()  
export class ExampleInterceptor implements HttpInterceptor {  
intercept(httpRequest: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
*//any alteration in httpRequest can be done here*  
return next.handle(httpRequest);  
}  
}

Here, Angular gives a reference to the httpRequest object when the intercept() function is used. We can review and alter this request as required. When our logic is finished, we use next.handle and send the revised request back to the application.

We'll also have to update it into the @NgModule

@NgModule({  
providers: [  
{   
provide: HTTP\_INTERCEPTORS, useClass: ExampleInterceptor, multi:true  
},  
],})

Operations of HTTP Interceptor

We can perform a wide range of actions using Angular interceptors. Let's take it one at a time:

Modify HTTP Headers

It is capable of modifying HTTP request header data.

@Injectable()  
export class HeaderInterceptor implements HttpInterceptor {  
constructor(private authService: AuthService) {}  
intercept(httpRequest: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
const reqCopy = req.clone()  
*//can set new header*  
reqCopy.headers.set({"ExampleHeader": "Test Data"})  
  
*//can modify header*  
reqCopy.headers.append({"Content-Type": "multipart/form-data"})  
  
*//can delete header*  
reqCopy.headers.delete("Content-Type")  
  
return next.handle(reqCopy);  
}  
}

Modifying The Request Body

The following steps can be taken to modify the request body:

@Injectable()  
export class RequestBodyInterceptor implements HttpInterceptor {  
intercept(httpRequest: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
const name = 'example';  
return next.handle(httpRequest.clone({ body: { "data":{...data, name: name }  
} }));  
}  
}

We can modify the request body contained within httpRequest.

Set Authentication/Authorization Token

In a real-time scenario, it may be necessary to include an API key or a JSON token in the header to allow authentication.

Using httpRequest.clone, we may clone a request and then forward it once it has been modified. As in the example, we will include authorization in the request header.

@Injectable()  
export class HeaderInterceptor implements HttpInterceptor {  
constructor(private authService: AuthService) {}  
intercept(httpRequest: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
const jwt = this.authService.getToken()  
return next.handle(httpRequest.clone({ setHeaders: { authorization: `Bearer ${jwt}` }   
}));  
}  
}

Modify The HTTP Response

The HTTP response can also be altered. If we want to transform XML data into JSON when it is received from the server, we may write the interceptor shown below:

@Injectable()  
export class HTTPResponseInterceptor implements HttpInterceptor {  
intercept(httpRequest: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
return next.handle(httpRequest).pipe(  
*// proceed when there is a response; ignore other events*  
filter(event => event instanceof HttpResponse),  
map(  
(event: HttpResponse<any>) => {  
  
*// {responseType: text} expects a string response*  
return event.clone({ body: JSON.stringify(this.xml.parse(event.body)) });   
})  
);  
}  
}

Here, we may add handler and subscribe to the httpResponse using pipe, and then read whenever it receives the response.

Error Handling

Since the interceptor can handle both the request and the response, you may write standard code to handle the HTTP error that you received as a response.

@Injectable()  
export class HttpResponseInterceptor implements HttpInterceptor {  
intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
return next.handle(request).pipe(  
tap({  
next: (event) => {  
if (event instanceof HttpResponse) {  
if(event.status == 401) {  
alert('Unauthorized access!')  
}  
}  
return event;  
},  
error: (error) => {  
if(error.status === 401) {  
alert('Unauthorized access!')  
}  
else if(error.status === 404) {  
alert('Page Not Found!')  
}  
}  
}));  
}  
}

Setting The New Headers

The following steps can be taken to add the new request header:

@Injectable()  
export class HeaderInterceptor implements HttpInterceptor {  
constructor(private authService: AuthService) {}  
intercept(httpRequest: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
const reqCopy = req.clone()  
*//can set new header*  
reqCopy.headers.set({"ExampleHeader": "Test Data"})  
  
return next.handle(reqCopy);  
}  
}

Cancel The Current Request

The following steps can be taken to cancel the current request:

import { EMPTY } from 'rxjs';  
  
@Injectable()  
export class HeaderInterceptor implements HttpInterceptor {  
constructor(private authService: AuthService) {}  
intercept(httpRequest: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
if (stopThisRequest) {  
return EMPTY;  
}  
  
return next.handle(reqCopy);  
}  
}

Change The Requested URL

Manipulating the URL seems unsafe, but let's see how easy it is in an interceptor.

It's as simple as cloning the request and changing http:// with [https://](https://inc-word-edit.officeapps.live.com/we/wordeditorframe.aspx?new=1&ui=en-US&rs=en-US&wopisrc=https%3A%2F%2Felipticoit-my.sharepoint.com%2Fpersonal%2Fharish_govindu_ispace_com%2F_vti_bin%2Fwopi.ashx%2Ffiles%2Fa423eb4b1e41456299867e3a505219ca&wdenableroaming=1&mscc=1&wdodb=1&hid=583C2EA1-60B7-3000-4DCA-37C72E72A822.0&uih=sharepointcom&wdlcid=en-US&jsapi=1&jsapiver=v2&corrid=69eed511-5622-315d-2047-7640fb0627c2&usid=69eed511-5622-315d-2047-7640fb0627c2&newsession=1&sftc=1&uihit=docaspx&muv=1&cac=1&sams=1&mtf=1&sfp=1&sdp=1&hch=1&hwfh=1&dchat=1&sc=%7B%22pmo%22%3A%22https%3A%2F%2Felipticoit-my.sharepoint.com%22%2C%22pmshare%22%3Atrue%7D&ctp=LeastProtected&rct=Normal&wdorigin=ONENOTE&wdredirectionreason=Unified_SingleFlush). The copied HTTPS request is then sent to the next handler.

@Injectable()  
export class ExampleInterceptor implements HttpInterceptor {  
intercept(httpRequest: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
return next.handle(req.clone({  
url: req.url.replace("http://", "[https://](https://inc-word-edit.officeapps.live.com/we/wordeditorframe.aspx?new=1&ui=en-US&rs=en-US&wopisrc=https%3A%2F%2Felipticoit-my.sharepoint.com%2Fpersonal%2Fharish_govindu_ispace_com%2F_vti_bin%2Fwopi.ashx%2Ffiles%2Fa423eb4b1e41456299867e3a505219ca&wdenableroaming=1&mscc=1&wdodb=1&hid=583C2EA1-60B7-3000-4DCA-37C72E72A822.0&uih=sharepointcom&wdlcid=en-US&jsapi=1&jsapiver=v2&corrid=69eed511-5622-315d-2047-7640fb0627c2&usid=69eed511-5622-315d-2047-7640fb0627c2&newsession=1&sftc=1&uihit=docaspx&muv=1&cac=1&sams=1&mtf=1&sfp=1&sdp=1&hch=1&hwfh=1&dchat=1&sc=%7B%22pmo%22%3A%22https%3A%2F%2Felipticoit-my.sharepoint.com%22%2C%22pmshare%22%3Atrue%7D&ctp=LeastProtected&rct=Normal&wdorigin=ONENOTE&wdredirectionreason=Unified_SingleFlush)")  
}));  
}  
}

This scenario may also be required if we need to add different API URLs for different development environments; this could be managed here.

Different routing stratagies

With client-side SPAs we have two strategies we can use to implement client-side routing, one is called the HashLocationStrategy and the other is called the PathLocationStrategy.

The default in Angular is the PathLocationStrategy, if we do nothing that is the strategy Angular will employ.

[Learning Objectives](https://codecraft.tv/courses/angular/routing/routing-strategies/#_learning_objectives)

* Know the difference between the HashLocationStrategy and PathLocationStrategy.
* Know the pros and cons of each and be able to make a judgement call for when to use each.

[HashLocationStrategy](https://codecraft.tv/courses/angular/routing/routing-strategies/#_hashlocationstrategy)

To enable HashLocationStrategy in an Angular application we pass {useHash: true} when we are providing our routes with RouterModule, like so:

TypeScript

CopyRouterModule.forRoot(routes, {useHash: true})

URL can contain some data prepended with a # character.

The # part of the URL is called the *hash fragment*.

It’s normally used so that people can link to a particular section in a HTML page, specifically anchor tags, for example if there is an anchor tag with an name attribute of routing-strategies like so:

HTML

Copy<a name="routing-strategies"></a>

Then if you created a bookmark of

Copyhttp://somedomain.com/page#routing-strategies

The browser would open somedomain.com/page and then *scroll down* so that the <a name="routing-strategies"></a> tag is at the top of the page.

However it has another very important characteristic in that anything past the # in a URL *never gets sent to the server*.

So if your URL was <https://codecraft.tv/contact/#/foo/moo/loo> then the browser makes a GET request to <https://codecraft.tv/contact/> **only**.

The #/foo/moo/loo part of the URL is **never** sent.

If you were to look at your logs on the server you would never see any reference to #/foo/moo/loo

Another way to think about the *hash fragment*, since it’s never sent to the server, is that it’s for storing the state of your *client* application.

It’s therefore an ideal solution for implementing *client* side routing:-

* It’s part of the URL so can be bookmarked and sent to other people.
* It won’t confuse the server side since the hash fragment is never sent to the server.
* It can be programmatically changed via JavaScript.

And that’s exactly why, for a number of years, the primary way of implementing client-side routing was via hash fragments.

Taking a look at the app we’ve built so far, if running locally the URLs look something like:

Copylocalhost:4040/#/search  
localhost:4040/#/artist/1234/tracks

According to the the server there is only ever one *URL* localhost:4040, the other *hash fragment* stuff is ignored by the server.

This is why we call what we are building a *Single Page Application*, there is only ever one *page* requested from the server. In the above example it’s localhost:4040 — the other pages are just changes to the hash fragment which the client application deals with, from the perspective of the server the whole site is just a single page.

[PathLocationStrategy](https://codecraft.tv/courses/angular/routing/routing-strategies/#_pathlocationstrategy)

This is the *default* strategy in Angular so we need to do nothing to enable it.

It takes advantage of a relatively new HTML5 API called pushstate (from the HTML5 history API).

By using pushstate we can change the URL and **not** have the browser request the page from the server and *without* needing to use a hash fragment.

So if we were at

Copylocalhost:4040/search

By using the pushstate API we can change the URL to

Copylocalhost:4040/artist/1234/tracks

And the browser **won’t** make a GET request to the server for /artist/1234/tracks.

That sounds perfect for client-side routing right?

* We get a URL that looks just like any other URL so can be bookmarked, shared and so on.
* The browser doesn’t send the request to the server so the routing is handled on the client side.

Unfortunately it has one big downside: if we reload the page, or bookmark it and open it later, the browser **would** make a request to the server for e.g. localhost:4040/artist/1234/tracks.

By using a hash fragment the server *never* needs to know about any application URL, it will only ever get asked for the root page and it will only ever return the root page.

But by using a PathLocationStrategy the server needs to be able to return the main application code for every URL, not just the root URL.

So with PathLocationStrategy we need to co-operate with a server side that supports this functionality, it’s possible and quite easy to implement a server side like this but it does require some effort and cooperation.

*Tip*

The local development server started by the Angular CLI *does* support this functionality so if you wanted to try it out you can.

[Base HRef](https://codecraft.tv/courses/angular/routing/routing-strategies/#_base_href)

When using the PathLocationStrategy we need to tell the browser what will be prefixed to the requested path to generate the URL.

We do that by specifying a base href, either in the head section of our HTML like so:

HTML

Copy<base href='/my/app'/>

Or you can *provide* it to the DI framework it via the symbol APP\_BASE\_HREF.

The value of the base href gets prepended to every navigation request we make, so if we ask to navigate to ['moo','foo'] with the above href the URL would change to /my/app/moo/foo

[Angular Universal](https://codecraft.tv/courses/angular/routing/routing-strategies/#_angular_universal)

What if we could build an app that rendered the page on the server side and returned that to the client, and from that point on let the client handle the routing?

What if we reloaded the page at localhost:4040/artist/1234/tracks and the server at localhost:4040 rendered the page?

The server could call the iTunes APIs and generate the HTML for the tracks page. It could return the page to the browser and the browser could just display it. Then if the user clicks on search the client application could take over and handle the routing on the client side.

That is something called *Angular Universal*, or *Isomorphic Rendering*, but essentially it’s the ability to run Angular in both the *browser* and the *server side*.

The big benefit of Angular Universal is that pages can be cached on the server side and applications will then load much faster.

For Angular Universal to work URLs need to be passed to the server side which is why it can only work with a PathLocationStrategy and not a HashLocationStrategy.

[Summary](https://codecraft.tv/courses/angular/routing/routing-strategies/#_summary)

The default client-side routing strategy used in Angular is the PathLocationStrategy.

This changes the URL programmatically using the HTML5 History API in such a way that the browser doesn’t make a request to the server for the new URL.

For this to work we do need to serve our Angular application from a server that supports requests on multiple different URLs, at a minimum all this server side needs to do is return the same page for all the different URLs that’s requested from it.

It’s not a lot of work but does need some co-operation from the server side.

PathLocationStrategy also sets us up for a future architecture where we can speed up loading time by pre-rendering the pages with Angular running on the server side and then once it’s downloaded to the browser the client can take over routing. This is called *Angular Universal* and it’s currently in development.

HashLocationStrategy uses the hash fragment part of the URL to store state for the client, it easier to setup and doesn’t require any co-operation from the server side but has the downside that it won’t work with *Angular Universal* once that’s released.

View Encapsulation

In Angular, **ViewEncapsulation** is a feature that controls the way styles are applied and scoped to components. It is a mechanism that encapsulates the styles defined in a component to prevent them from affecting other components in the application.

By default, Angular uses the **ViewEncapsulation.Emulated** mode, also known as the **“shadow DOM”** emulation. In this mode, Angular emulates the behavior of the shadow DOM by adding a unique attribute to the component’s HTML elements and applying styles with these attributes. This way, styles defined in a component only affect the elements within that component’s template and do not leak out to other parts of the application.

**There are three ViewEncapsulation modes available in Angular:**

* 1. **ViewEncapsulation.Emulated (default):** This mode emulates the shadow DOM by adding unique attributes to the component’s elements. The styles defined within the component’s template are scoped to that component only. **ViewEncapsulation.Emulated** will add the css style in the head section of your website and reference your component’s unique id(\_ngcontent) to apply it.
  2. **ViewEncapsulation.None:** In this mode, styles defined in a component’s template are not encapsulated and can affect the entire application. It’s important to use this mode with caution, as it can lead to style collisions and unintended side effects when multiple components use the same styles.
  3. **ViewEncapsulation.ShadowDom:** This mode uses the native browser’s shadow DOM implementation to encapsulate the styles. It requires the browser to support the shadow DOM. With this mode, the component’s styles are truly isolated within the component and do not leak out to other components or the global styles. **ViewEncapsulation.ShadowDom** will add the css style inside the generated DOM of your component.

To specify the ViewEncapsulation mode for a component, you can use the **`encapsulation`** property in the component’s metadata:

import { Component, ViewEncapsulation } from ‘@angular/core’;  
  
@Component({  
selector: ‘app-my-component’,  
templateUrl: ‘./my-component.component.html’,  
styleUrls: [‘./my-component.component.css’],  
encapsulation: ViewEncapsulation.Emulated // or ViewEncapsulation.None or ViewEncapsulation.ShadowDom  
})  
export class MyComponent {  
// Component logic goes here  
}

**Conclusion:**

By understanding and utilizing **ViewEncapsulation** in Angular, you can have better control over styles and prevent unintended style interference between components in your application.

How to handle multiple http requests in Angular?

When dealing with multiple HTTP requests in Angular, you have several options. Let’s explore a few approaches:

1. Sequential Requests with Nested Subscriptions: You can chain HTTP requests sequentially using nested subscriptions. While this approach works, it’s not the most efficient or readable way. Here’s an example:

import { Component } from '@angular/core';

import { HttpClient } from '@angular/common/http';

@Component({

selector: 'app-root',

templateUrl: 'app/app.component.html'

})

export class AppComponent {

loadedCharacter: {};

constructor(private http: HttpClient) {}

ngOnInit() {

this.http.get('/api/people/1').subscribe(character => {

this.http.get(character.homeworld).subscribe(homeworld => {

character.homeworld = homeworld;

this.loadedCharacter = character;

});

});

}

}

[While this code works, it can become unwieldy when dealing with more requests](https://stackoverflow.com/questions/64822065/best-way-for-multiple-http-request-in-angular)

Using forkJoin: The forkJoin operator allows you to wait for multiple observables to complete and then combine their results. It runs all requests concurrently and waits for all of them to finish. Here’s how you can use it:

import { Component } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { forkJoin } from 'rxjs';

@Component({

selector: 'app-root',

templateUrl: 'app/app.component.html'

})

export class AppComponent {

loadedCharacter: {};

constructor(private http: HttpClient) {}

ngOnInit() {

const characterRequest = this.http.get('/api/people/1');

const homeworldRequest = this.http.get('/api/planets/1');

forkJoin([characterRequest, homeworldRequest]).subscribe(([character, homeworld]) => {

character.homeworld = homeworld;

this.loadedCharacter = character;

});

}

}

[This approach is more concise and efficient, especially when dealing with multiple requests2](https://stackoverflow.com/questions/45117273/how-to-use-multiple-http-requests-in-angular-4).

Using zip Operator: The zip operator combines multiple observables into one observable that emits an array of values. It executes all requests in parallel and waits for all of them to complete. Here’s an example:

import { Component } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { zip } from 'rxjs';

@Component({

selector: 'app-root',

templateUrl: 'app/app.component.html'

})

export class AppComponent {

loadedCharacter: {};

constructor(private http: HttpClient) {}

ngOnInit() {

const characterRequest = this.http.get('/api/people/1');

const homeworldRequest = this.http.get('/api/planets/1');

zip(characterRequest, homeworldRequest).subscribe(([character, homeworld]) => {

character.homeworld = homeworld;

this.loadedCharacter = character;

});

}

}

[The zip operator is useful when you want to execute requests concurrently and handle their results together3](https://stackoverflow.com/questions/66351564/angular-rxjs-handle-multiple-http-requests).

Error Handling: Remember to handle errors appropriately. If any request fails, you can catch errors using the .pipe() operator and handle them accordingly.

Choose the approach that best fits your use case and keeps your code clean and maintainable. [Happy coding! 😊4](https://medium.com/@swarnakishore/performing-multiple-http-requests-in-angular-4-5-with-forkjoin-74f3ac166d61)[5](https://coryrylan.com/blog/angular-multiple-http-requests-with-rxjs)

Feel free to explore these options and adapt them to your specific requirements! If you have any more questions, feel free to ask.

2.What is the difference between forkJoin and zip?

Certainly! Let’s explore the differences between forkJoin and zip in the context of RxJS observables:

* 1. **forkJoin**:

Purpose: forkJoin is used to combine multiple observables into a single observable that emits an array of values. It waits for all the source observables to complete and then emits an array containing the last value from each observable.

Concurrency: All source observables run concurrently, and forkJoin waits for all of them to complete.

Error Handling: If any of the source observables encounter an error, the entire combined observable will also emit an error.

Example:

**import { forkJoin } from 'rxjs';**

**const observable1 = ...; // First observable**

**const observable2 = ...; // Second observable**

**forkJoin([observable1, observable2]).subscribe(([value1, value2]) => {**

**// Handle the combined results**

**});**

* 1. zip:
  + **Purpose:** zip combines multiple observables into one observable that emits an array of values, where each value corresponds to the nth value emitted by each source observable. It runs all source observables in parallel.
  + **Concurrency:** Observables run concurrently, but the emitted values are paired based on their order (i.e., the first value from each observable is combined, the second value from each observable, and so on).
  + **Error Handling:** If any of the source observables encounter an error, the combined observable will emit an error, but it will continue to emit values for other observables.

**import { zip } from 'rxjs';**

**const observable1 = ...; // First observable**

**const observable2 = ...; // Second observable**

**zip(observable1, observable2).subscribe(([value1, value2]) => {**

**// Handle the combined results**

**});**

* 1. Use Cases:
     + Use forkJoin when you need to wait for all observables to complete and collect their final values together.
     + Use zip when you want to combine values from multiple observables in parallel, regardless of their completion order.

Remember to choose the operator that best fits your specific use case! If you have any more questions or need further clarification, feel free to ask. 😊

Can you show an example of parallel requests using mergeMap?

Certainly! Let’s explore an example of how to handle parallel requests using the mergeMap operator in Angular with RxJS.

Suppose we have two independent API requests that we want to execute concurrently. We’ll use the mergeMap operator to achieve this. First, let’s set up a scenario where we fetch data from two different endpoints:

* 1. Fetch a single todo item from an API.
  2. Fetch comments related to that todo item.

Here’s how you can use mergeMap to handle these parallel requests:

**TypeScript**

import { HttpClient } from '@angular/common/http';

import { Component, OnInit } from '@angular/core';

import { mergeMap } from 'rxjs/operators';

@Component({

selector: 'app-parallel-requests',

templateUrl: './parallel-requests.component.html',

})

export class ParallelRequestsComponent implements OnInit {

todoValue: string = '';

comments: any[] = [];

constructor(private http: HttpClient) {}

ngOnInit(): void {

// Fetch a single todo item

this.http.get<any>('https://jsonplaceholder.typicode.com/todos/1').pipe(

mergeMap((todo: any) => {

this.todoValue = todo.title;

// Fetch comments related to the todo item

return this.http.get<any[]>(`https://jsonplaceholder.typicode.com/posts/${todo.id}/comments`);

})

).subscribe((comments: any[]) => {

this.comments = comments;

});

}

}

In this example:

* 1. We start by fetching a single todo item using the first API call.
  2. The mergeMap operator then takes the result of the first request (the todo item) and maps it to the second API call (fetching comments).
  3. Both requests run concurrently, and once both complete, the comments are assigned to the comments property.

Remember that mergeMap is useful when you have multiple parallel API requests that are independent of each other and you want to wait for all of them to complete before taking further action. It helps avoid nested subscriptions and provides a streamlined approach to handling asynchronous requests.

What are some other RxJS operators for combining observables?

Certainly! Besides merge, there are several other RxJS operators for combining observables. Let’s explore a few of them:

1. concat:
   * The concat operator combines multiple observables sequentially. It waits for the first observable to complete before subscribing to the next one. Think of it as appending one observable’s emissions to the end of another.

import { concat, interval } from 'rxjs';

const first = interval(1000);

const second = interval(1500);

const combined = concat(first, second);

// Output: 0, 1, 2, 3, 4, 0, 1, 2, 3, 4, ...

combined.subscribe(val => console.log(val));

1. combineLatest:
   * The combineLatest operator combines the latest values from multiple observables. It emits an array of the most recent values whenever any of the source observables emit a new value.

import { combineLatest, interval } from 'rxjs';

const first = interval(1000);

const second = interval(1500);

const combined = combineLatest(first, second);

// Output: [0, 0], [1, 0], [1, 1], [2, 1], [2, 2], ...

combined.subscribe(val => console.log(val));

1. zip:
   * The zip operator combines multiple observables by pairing their emissions based on their order. It emits an array containing the nth value from each observable.

import { zip, interval } from 'rxjs';

const first = interval(1000);

const second = interval(1500);

const combined = zip(first, second);

// Output: [0, 0], [1, 1], [2, 2], [3, 3], ...

combined.subscribe(val => console.log(val));

1. ForkJoin:
   * The forkJoin operator waits for all source observables to complete and then emits an array containing the last value from each observable. It’s useful for combining observables that complete at different times.

import { forkJoin, of } from 'rxjs';

const first = of('Hello');

const second = of('World');

forkJoin([first, second]).subscribe(val => console.log(val)); // Output: ['Hello', 'World']

Remember to choose the operator that best suits your use case based on the behavior you need. [Each operator has its own characteristics, so pick the one that aligns with your specific requirements!](https://www.learnrxjs.io/learn-rxjs/operators/combination/merge)

What is the difference between mergeMap and switchMap?

Certainly! Let’s explore the difference between mergeMap and switchMap in the context of RxJS observables:

* 1. mergeMap (also known as flatMap):
  + **Behavior:**
    - mergeMap flattens an observable by merging the inner observables into a single observable stream.
    - It subscribes to all inner observables concurrently and emits their values as they arrive.
    - It maintains all active inner subscriptions, allowing multiple inner observables to emit values simultaneously.
  + **Use Case:**
    - Use mergeMap when you want to process multiple observables concurrently and collect their results.
    - Example: Imagine fetching data from multiple endpoints simultaneously and combining the results into a single stream.

import { of } from 'rxjs';

import { mergeMap } from 'rxjs/operators';

const source = of(1, 2, 3);

source.pipe(

mergeMap((value) => of(value \* 10))

).subscribe((result) => {

console.log(result); // Output: 10, 20, 30

});

* 1. switchMap:
     + switchMap flattens an observable by canceling the previous inner observable when a new outer value arrives.

It subscribes to the latest inner observable and discards any ongoing inner

subscriptions.

* + - Useful for scenarios where you want to switch to a new inner observable whenever the outer observable emits a new value.

**Use Case:**

Use switchMap when you need to handle ongoing requests (e.g., HTTP calls) and cancel previous requests when a new one is triggered.

* + - * Example: Autocomplete search suggestions where you want to cancel previous search requests when the user types a new query.

import { of, interval } from 'rxjs';

import { switchMap } from 'rxjs/operators';

const source = interval(1000); // Emits values 0, 1, 2, ...

source.pipe(

switchMap((value) => of(value \* 10))

).subscribe((result) => {

console.log(result); // Output: 0, 10, 20, 30, ...

});

In summary:

* + - mergeMap maintains all inner subscriptions and combines their emissions.
    - switchMap cancels previous inner subscriptions and switches to the latest inner observable.

Choose the operator that best suits your use case based on whether you want to merge or switch between inner observables

ForkJoin-Can you show an example of parallel requests using forkJoin?

Certainly! When you have multiple parallel API requests that are independent of each other and you want to wait for all of them to complete before taking further action, you can use the forkJoin operator. It allows you to combine multiple observables and execute them concurrently. Once all the observables emit their values, forkJoin emits a single observable value containing an array of all the resolved values.

Let’s explore an example where we want to load character data and their homeworld concurrently. We’ll use the Star Wars API (SWAPI) to demonstrate this:

import { HttpClient } from '@angular/common/http';

import { Component, OnInit } from '@angular/core';

import { forkJoin } from 'rxjs';

@Component({

selector: 'app-character-details',

templateUrl: './character-details.component.html',

})

export class CharacterDetailsComponent implements OnInit {

character: any;

homeworld: any;

constructor(private http: HttpClient) {}

ngOnInit(): void {

// Fetch character details

const characterRequest = this.http.get<any>('https://swapi.dev/api/people/1');

// Fetch character's homeworld

const homeworldRequest = this.http.get<any>('https://swapi.dev/api/planets/1');

// Combine both requests using forkJoin

forkJoin([characterRequest, homeworldRequest]).subscribe(

([character, homeworld]) => {

this.character = character;

this.homeworld = homeworld;

console.log('Character:', character.name);

console.log('Homeworld:', homeworld.name);

},

(error) => {

console.error('Error fetching data:', error);

}

);

}

}

In this example:

* + We make two API requests concurrently: one for character details and another for the character’s homeworld.
  + The forkJoin operator waits for both requests to complete.
  + Once both requests are successful, it emits an array containing the character details and homeworld data.

Remember to replace the API endpoints with your actual API URLs. forkJoin is a powerful tool for handling parallel requests efficiently. [Feel free to adapt this example to your specific use case! 😊 1](https://coryrylan.com/blog/angular-multiple-http-requests-with-rxjs)

ConcatMap- Can you show an example of sequential requests using concatMap?

Certainly! When you need to make a series of HTTP requests sequentially, where each subsequent request depends on the result of the previous one, you can use the concatMap operator from RxJS. Let’s explore an example:

Suppose you have a list of user IDs, and you want to fetch user details for each ID sequentially. Here’s how you can achieve this using concatMap:

import { from } from 'rxjs';

import { concatMap } from 'rxjs/operators';

const userIds = [1, 2, 3, 4, 5];

from(userIds).pipe(

concatMap((userId) => getUserDetails(userId))

).subscribe(

(user) => {

console.log(`User ${user.id}: ${user.name}`);

},

(error) => {

console.error('Error fetching user details:', error);

}

);

// Simulate fetching user details from an API

function getUserDetails(userId: number) {

// Replace with your actual API call

return fetch(`https://api.example.com/users/${userId}`)

.then((response) => response.json());

}

In this example:

* + We start with an array of user IDs (userIds).
  + The from(userIds) creates an observable that emits each user ID sequentially.
  + The concatMap operator takes each user ID, fetches the user details using the getUserDetails function (simulated here), and emits the result.
  + The requests are executed one after the other, ensuring sequential execution.

Remember to replace the getUserDetails function with your actual API call. This approach ensures that each request waits for the previous one to complete before proceeding.

Feel free to adapt this example to your specific use case! [If you have any more questions or need further clarification, feel free to ask](https://stackoverflow.com/questions/53560652/how-to-make-a-sequence-of-http-requests-in-angular-6-using-rxjs)

Performance improvement in the Angular application?

* Remove unnecessary library and code
* Use trackBy in ngFor loops
* Caching the static file to improve the performance
* ng-container is a structural directive that doesn't create any additional DOM elements. ng-container is to provide a grouping mechanism for applying structural directives like ngIf, ngFor, and ngSwitch without introducing unnecessary HTML elements.
* Compressed the image size in application
* Implement lazy loading for modules. This helps in loading only the necessary parts of your application when they are needed, reducing the initial load time
* Use pagination or infinite scrolling to improve the data load in the page.
* OnPush change detection strategy is also used for improving the Angular performance improvement. Angular doesn’t need to traverse the entire component tree structure for detecting individual changes on properties. We can re render the component on demand in this strategy.

Angular State Management

State management in Angular involves managing the data and user interface (UI) states within your application. This is especially important in large and complex applications where components need to share and synchronize data. There are various state management techniques in Angular, but we'll focus on the following three popular approaches:

* 1. Component State
  2. Services and RxJS
  3. NgRx Store

Let's dive into each of these options and see how they work.

Component State

Component state is the simplest form of state management in Angular.

It involves storing data within individual components.

Each component has its own state, and data is passed down through the component hierarchy using input and output bindings.

This approach is suitable for small to moderately complex applications where data doesn't need to be shared extensively.'

Here's a simple example of a component state in Angular:

In this example, the AppComponent has its own state (title and message), and it passes the message down to the AppChildComponent. When the button is clicked in the child component, it emits an event that updates the message in the parent component.

// app.component.ts import { Component } from '@angular/core'; @Component({

selector: 'app-root',

template: ` <h1>{{ title }}</h1> <app-child [message]="message" (updateMessage)="updateMessage($event)"></app-child> ` })

export class AppComponent {

title = 'Component State Example';

message = 'Hello from AppComponent!'; updateMessage(newMessage: string) {

this.message = newMessage;

}

}

// app-child.component.ts import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({

selector: 'app-child',

template: ` <div>{{ message }}</div> <button (click)="changeMessage()">Change Message</button> ` })

export class AppChildComponent {

@Input() message: string;

@Output() updateMessage = new EventEmitter<string>(); changeMessage()

{

const newMessage = 'Updated message from AppChildComponent!'; this.updateMessage.emit(newMessage);

}

}

Component state is a straightforward approach, but it has limitations when it comes to sharing data across unrelated components or managing asynchronous operations.

**Services and RxJS**

Services and RxJS (Reactive Extensions for JavaScript) are often used together for state management in Angular applications. Services act as centralized stores for application data, and RxJS provides a powerful way to handle asynchronous operations and data streams.

Here's an example of using services and RxJS for state management:

// data.service.ts  
import { Injectable } from '@angular/core';  
import { BehaviorSubject } from 'rxjs';  
  
@Injectable({  
 providedIn: 'root'  
})  
export class DataService {  
 private messageSubject = new BehaviorSubject<string>('Hello from DataService');  
 message$ = this.messageSubject.asObservable();  
  
 updateMessage(newMessage: string) {  
 this.messageSubject.next(newMessage);  
 }  
}  
// app.component.ts  
import { Component } from '@angular/core';  
import { DataService } from './data.service';  
  
@Component({  
 selector: 'app-root',  
 template: `  
 <h1>{{ title }}</h1>  
 <div>{{ message$ | async }}</div>  
 <button (click)="changeMessage()">Change Message</button>  
 `  
})  
export class AppComponent {  
 title = 'Services and RxJS Example';  
  
 constructor(private dataService: DataService) {}  
  
 changeMessage() {  
 const newMessage = 'Updated message from AppComponent!';  
 this.dataService.updateMessage(newMessage);  
 }  
}  
// app-child.component.ts  
import { Component } from '@angular/core';  
import { DataService } from './data.service';  
  
@Component({  
 selector: 'app-child',  
 template: `  
 <div>{{ message$ | async }}</div>  
 `  
})  
export class AppChildComponent {  
 constructor(private dataService: DataService) {}  
}

In this example, we have a DataService that uses a BehaviorSubject to store and emit the message. Both the AppComponent and AppChildComponent subscribe to the message$ observable to receive updates when the message changes. This approach allows for efficient data sharing and synchronization across components.

[Services and RxJS](https://hackernoon.com/building-your-own-custom-pipable-rxjs-operator?ref=hackernoon.com) are a flexible and powerful way to manage state in Angular, especially for medium to large-scale applications with complex data flows.

NgRx Store

NgRx is a state management library inspired by Redux for Angular applications. It introduces a unidirectional data flow and a centralized store for managing the application state. NgRx Store is most suitable for large and complex applications where state management becomes critical.

Let's see an example of using NgRx Store for state management:

// app.state.ts  
import { createAction, createReducer, on, createSelector } from '@ngrx/store';  
  
export interface AppState {  
 message: string;  
}  
  
export const initialAppState: AppState = {  
 message: 'Hello from NgRx Store'  
};  
  
export const updateMessage = createAction(  
 '[App] Update Message',  
 (newMessage: string) => ({ newMessage })  
);  
  
export const appReducer = createReducer(  
 initialAppState,  
 on(updateMessage, (state, { newMessage }) => ({ ...state, message: newMessage }))  
);  
  
// Selector  
export const selectMessage = (state: AppState) => state.message;  
export const getMessage = createSelector(selectMessage, (message) => message);  
// app.component.ts  
import { Component } from '@angular/core';  
import { Store } from '@ngrx/store';  
import { updateMessage, getMessage } from './app.state';  
  
@Component({  
 selector: 'app-root',  
 template: `  
 <h1>{{ title }}</h1>  
 <div>{{ message$ | async }}</div>  
 <button (click)="changeMessage()">Change Message</button>  
 `  
})  
export class AppComponent {  
 title = 'NgRx Store Example';  
 message$ = this.store.select(getMessage);  
  
 constructor(private store: Store) {}  
  
 changeMessage() {  
 const newMessage = 'Updated message from AppComponent!';  
 this.store.dispatch(updateMessage({ newMessage }));  
 }  
}  
// app-child.component.ts  
import { Component } from '@angular/core';  
import { Store } from '@ngrx/store';  
import { getMessage } from './app.state';  
  
@Component({  
 selector: 'app-child',  
 template: `  
 <div>{{ message$ | async }}</div>  
 `  
})  
export class AppChildComponent {  
 message$ = this.store.select(getMessage);  
  
 constructor(private store: Store) {}  
}

In this example, we define an AppState interface, actions, reducers, and selectors to manage the state. Both the AppComponent and AppChildComponent use the NgRx Store to select and update

the message. NgRx Store provides a structured and scalable way to handle state in large [Angular applications](https://hackernoon.com/testing-angular-apps-with-cypress?ref=hackernoon.com).

Conclusion

In Angular, state management is a crucial aspect of building scalable and maintainable applications. The choice of state management approach depends on the complexity of your application and your specific requirements. Here's a summary of when to use each approach:

* Component State: Suitable for small to moderately complex applications with limited data sharing needs.
* Services and RxJS: Ideal for medium to large-scale applications with asynchronous data and moderate complexity.
* NgRx Store: Best suited for large and complex applications with extensive data sharing and state management requirements.

By understanding and choosing the right state management approach for your Angular project, you can ensure better organization, maintainability, and scalability of your application's codebase. Remember that the MECE (Mutually Exclusive, Collectively Exhaustive) principle applies when selecting these options, and it's often beneficial to combine them to meet your specific needs.

Pre- Rendering

Pre-rendering in Angular 9

[**What is Pre-rendering in angular?**](https://www.thirdrocktechkno.com/blog/prerendering-in-angular-9/#what-is-pre-rendering-in-angular)

[**Why should we pre-render Angular applications?**](https://www.thirdrocktechkno.com/blog/prerendering-in-angular-9/#why-should-we-pre-render-angular-applications)

[**How to implement pre-render Angular applications?**](https://www.thirdrocktechkno.com/blog/prerendering-in-angular-9/#how-to-implement-pre-render-angular-applications)

[**Conclusion**](https://www.thirdrocktechkno.com/blog/prerendering-in-angular-9/#conclusion)

What is Pre-rendering in angular?

Pre-rendering creates a static page when we [**build an angular application**](https://www.thirdrocktechkno.com/hire-us/hire-web-developer/hire-angular-developer/)**. As a result, the website will load faster and will be SEO-friendly without the need for** [**server-side rendering**](https://www.thirdrocktechkno.com/blog/server-side-rendering-with-angular-10/)**. The pre-rendering is helpful if we want to serve static pages in the application.**

Why should we pre-render Angular applications?

Most of the obstacles that one faces with an [Angular application](https://www.thirdrocktechkno.com/services/web-development/angular-development/) are fixed by following the optimization techniques. Although there are still a few problems that optimizations won’t fix:

* 1. SEO (search engine optimization): SPAs ([single-page applications](https://www.thirdrocktechkno.com/blog/single-page-apps-vs-multi-page-apps-what-to-choose-for-web-development)) are harder to index by search engines because the content isn’t available on load time. Therefore, the application is likely to fail on several SEO requirements.
  2. Slow initial page load: Since the application still needs to be bootstrapped after the page is loaded, there is an initial waiting time until the user can use the application. This results in a bad user experience.

Now, we have two ways to solve this problem.

* 1. Server-Side Rendering (SSR): SSR executes the [Angular application](https://www.thirdrocktechkno.com/blog/single-page-apps-vs-multi-page-apps-what-to-choose-for-web-development) on the server. The server will serve the compiled content in a way that search engine crawlers can read it. SSR is the best of both worlds. The server will render the application, but when the JavaScript bundle is loaded, it will turn into a SPA. The result is an application with rich UI/UX that loads quickly at the same time!
  2. Pre-Rendering: Pre-rendering would run generated static HTML files when we create build, and that is insanely fast. And when the JavaScript bundles are loaded, the browser would take over. What we get after this is Ultra fast loading time + No compromise in the rich [SPA experience](https://www.thirdrocktechkno.com/blog/angular-web-development-why-build-a-single-page-application/).

Both the technique solves the issue we have above, but if you see the rendering time that we have observed, it will make sense to choose the later.

Updates in Angular 9

How to implement pre-render Angular applications?

First, you need to add angular universal to your project using this command from the server-side rendering guide

Must use Node 12 or above Version

**ng add @nguniversal/express-engine --clientProject project-example**

The command will generate all the scripts to run server-side rendering in our application. You need it to generate static pages.

The next step is to transfer the code fragments responsible for creating the express server from server.ts to a new express-app.ts file. Below is the code that you should paste into the new file.

Once you copy the code remove the express server code from the server.ts and instead, import it from the express-app.ts leaving only the code listening to the port.

Server Side Rendering with Angular 10

[Read More](https://www.thirdrocktechkno.com/blog/server-side-rendering-with-angular-10/)

// server.ts  
  
  
import { app } from './express-app';  
  
const PORT = process.env.PORT || 4000;  
  
// Start up the Node server  
app.listen(PORT, async () => {  
console.log(`Node Express server listening on http://localhost:${PORT}`);  
});

Further, now add the main code that performs pre-render of our sub-pages. (you also need to install an additional library to create sub-folders using the command npm install mkdirp @types/mkdirp)

// Prerender.ts  
  
import \* as request from 'request-promise';  
import \* as mkdirp from 'mkdirp';  
import { promisify } from 'util';  
import { writeFileSync } from 'fs';  
import { Express } from 'express';  
  
import { app } from './express-app';  
  
export const ROUTES = [  
'/',  
'/auth',  
'/privacy-policy'  
];  
  
const mkdirpAsync = promisify(mkdirp);  
  
function prerender(expressApp: Express, routes: string[]) {  
const PORT = process.env.PRERENDER\_PORT || 4000;  
// Start up the Node server  
const server = expressApp.listen(PORT, async () => {  
try {  
for (const route of routes) {  
const result = await request.get(`http://localhost:${PORT}${route}`);  
await mkdirpAsync(`dist/browser${route}`);  
writeFileSync(`dist/browser${route}/index.html`, result);  
}  
console.log('Prerender complete.');  
server.close();  
} catch (error) {  
server.close(() => process.exit(1));  
}  
});  
}  
  
prerender(app, ROUTES);// Work around Solution for <https://github.com/angular/angular-cli/issues/7200> (This is just for reference)  
  
const path = require('path');  
const webpack = require('webpack');  
  
module.exports = {  
mode: 'none',  
entry: {  
// This is our Express server for Dynamic universal  
server: './server.ts',  
prerender: './prerender.ts' // <--------------- HERE!!!  
},  
externals: {  
'./dist/server/main': 'require("./server/main")'  
},  
target: 'node',  
resolve: { extensions: ['.ts', '.js'] },  
optimization: {  
minimize: false  
},  
output: {  
// Puts the output at the root of the dist folder  
path: path.join(\_\_dirname, 'dist'),  
filename: '[name].js'  
},  
module: {  
noParse: /polyfills-.\*\.js/,  
rules: [  
{ test: /\.ts$/, loader: 'ts-loader' },  
{  
// Mark files inside `@angular/core` as using SystemJS style dynamic imports.  
// Removing this will cause deprecation warnings to appear.  
test: [/(\\|\/)@angular(\\|\/)core(\\|\/).+\.js$/](mailto:/(\\%7C\/)@angular(\\%7C\/)core(\\%7C\/).+\.js$/),  
parser: { system: true }  
}  
]  
},  
plugins: [  
new webpack.ContextReplacementPlugin(  
// fixes WARNING Critical dependency: the request of a dependency is an expression  
/(.+)?angular(\\|\/)core(.+)?/,  
path.join(\_\_dirname, 'src'), // location of your src  
{} // a map of your routes  
),  
new webpack.ContextReplacementPlugin(  
// fixes WARNING Critical dependency: the request of a dependency is an expression  
/(.+)?express(\\|\/)(.+)?/,  
path.join(\_\_dirname, 'src'),  
{}  
)  
]  
};

[Read More](https://www.thirdrocktechkno.com/blog/why-hire-an-angularjs-development-company-for-app-development/)

You will need to add a new script to package.json for ease of use.

"scripts": {  
...  
"prerender": "node dist/prerender",  
...  
}

And that’s all, now you just need to build an application in SSR mode and run the command "prerender".

npm run build:ssr && npm run prerender

In the dist/browser directory, you will find sub-folders with index.html files containing SEO-friendly generated HTML content of the application.

If you are using @angular/service-worker then you will need to reconfigure it right after the prerender because the checksum value in the index.html file has changed after the HTML modification. Use the command below to do so.

ngsw-config dist/browser ngsw-config.json

**Conclusion**

Pre-render allows us to generate HTML files for every route so that we can serve content faster.

To ensure that clients can only download the files that they are permitted to see, put all client-facing asset files in the /dist folder and only honor requests for files from the /dist folder.

Pre-Rendering vs Server side rendering

Certainly! Let’s dive into the differences between **Server-Side Rendering (SSR)** and **Pre-Rendering** in Angular:

1. **Server-Side Rendering (SSR)**:
   * SSR involves rendering your Angular application on the server side before sending it to the client’s browser.
   * The server generates fully rendered HTML pages, which can be parsed and displayed even before the application JavaScript is downloaded.
   * Advantages of SSR:
     + **Improved Performance**: By delivering pre-rendered HTML, SSR reduces the time to first meaningful paint, enhancing performance.
     + **SEO Benefits**: Search engines can easily crawl and index the content since it’s available in HTML format.
   * Use cases for SSR:
     + When your content changes frequently and you need dynamic data.
     + [For pages with dynamic links, especially when dealing with IDs1](https://angular.io/guide/ssr?ref=workingsoftware-dev).
     1. **Pre-Rendering**:
        + Pre-rendering creates a set of static HTML pages during build time that can be directly used for client browsers.
        + It’s suitable for pages that don’t change frequently, such as contact, about, and landing pages.
        + Advantages of Pre-Rendering:
          - **Blazing-Fast Page Loads**: Since the HTML is already generated, the initial page load is lightning-fast.
          - **SEO Benefits**: Like SSR, pre-rendered pages are easily indexed by search engines.
        + Use cases for Pre-Rendering:
          - [Primarily static pages where content remains relatively constant](https://angular.io/guide/ssr?ref=workingsoftware-dev)[2](https://medium.com/geekculture/angular-server-side-rendering-ssr-vs-pre-rendering-and-why-do-it-in-the-first-place-28419e3220db)[3](https://medium.com/@deyan.p.peychev/unraveling-the-web-server-side-rendering-vs-pre-rendering-in-angular-universal-78c0cb8b9919).

In summary:

* + - Use **pre-rendering** for static pages.
    - Use **SSR** for dynamic pages or when dealing with changing content.

Remember, the choice between SSR and pre-rendering depends on your specific use case and requirements! [😊🚀](https://angular.io/guide/ssr?ref=workingsoftware-dev)[4](https://www.digitalocean.com/community/tutorials/angular-angular-universal)

Change detection

Change Detection means updating the DOM whenever data is changed. Angular provides two strategies for Change Detection.

In its **default** strategy, whenever any data is mutated or changed, Angular will run the change detector to update the DOM.

In the **onPush** strategy, Angular will only run the change detector when a new reference is passed to @Input() data.

To update the DOM with updated data, Angular provides its own change detector to each component, which is responsible for detecting change and updating the DOM.

Let's say we have a MessageComponent, as listed below:

import { Component, Input } from '@angular/core';

@Component({

selector: 'app-message',

template: `

<h2>

Hey {{person.firstname}} {{person.lastname}} !

</h2>

`

})

export class MessageComponent {

@Input() person;

}

In addition, we are using MessageComponent inside AppComponent as shown below:

import { Component, OnInit } from '@angular/core';

@Component({

selector: 'app-root',

template: `

<app-message [person]='p'></app-message>

<button (click)='changeName()'>Change Name</button>

`

})

export class AppComponent implements OnInit {

p: any;

ngOnInit(): void {

this.p = {

firstname: 'Brad',

lastname: 'Cooper'

};

}

}

* + Let us talk through the code: all we are doing is using MessageComponent as a child inside AppComponent and setting the value of person using the property binding. At this point in running the application, you will get the name printed as output.
  + Next, let's go ahead and update the firstname property on the button click in the AppComponent class below:

changeName() {

this.p.firstname = 'Foo';

}

As soon as we changed the property of mutable object P, Angular fires the change detector to make sure that the DOM (or view) is in sync with the model (in this case, object p). For each property changes, Angular change detector will traverse the component tree and update the DOM.

Let's start with understanding the component tree. An Angular application can be seen as a component tree. It starts with a root component and then goes through to the child components. In Angular, data flows from top to bottom in the component tree.

* 1. Whenever the @Input type property will be changed, the Angular change detector will start form the root component and traverse all child components to update the DOM.
  2. Any changes in the primitive type's property will cause Angular change detection to detect the change and update the DOM.
  3. In the above code snippet, you will find that on click of the button, the first name in the model will be changed.
  4. Then, change detection will be fired to traverse from root to bottom to update the view in MessageComponent.

There could be various reasons for Angular change detector to come into action and start traversing the component tree. They are:

* 1. Events fired such as button click, etc.
  2. AJAX call or XHR requests.
  3. Use of JavaScript timer functions such as setTimeOut , SetInterval.
  + Now, as you see, a single property change can cause change detector to traverse through the whole component tree.
  + Traversing and change detection is a heavy process, which may cause performance degradation of application.
  + Imagine that there are thousands of components in the tree and mutation of any data property can cause change detector to traverse all thousand components to update the DOM.
  + To avoid this, there could be a scenario when you may want to instruct Angular that when change detector should run for a component and its subtree, you can instruct a component's change detector to run only when object references changes instead of mutation of any property by choosing the **onPushChangeDetection** strategy.

**You may wish to instruct Angular to run change detection on components and their sub-tree only when new references are passed to them versus when data is simply mutated by setting change detection strategy to onPush.**

* + Let us go back to our example where we are passing an object to MessageComponent.
  + In the last example, we just changed the firstname property and that causes change detector to run and to update the view of MessageComponent.
  + However, now we want change detector to only run when the reference of the passed object is changed instead of just a property value.
  + To do that, let us modify MessageComponent to use the OnPush ChangeDetection strategy. To do this set the changeDetection property of the @Component decorator to ChangeDetectionStrategy.OnPush as shown in listing below:

import { Component, Input, ChangeDetectionStrategy } from '@angular/core';

@Component({

selector: 'app-message',

changeDetection: ChangeDetectionStrategy.OnPush,

template: `

<h2>

Hey {{person.firstname}} {{person.lastname}} !

</h2>

`

})

export class MessageComponent {

@Input() person;

}

* + At this point when you run the application, on the click event of the button in the AppComponent change detector will not run for MessageComponent, as only a property is being changed and reference is not changing.
  + Since the change detection strategy is set to onPush, now the change detector will only run when the reference of the @Input property is changed.

changeName() {

this.p = {

firstname: 'Foo',

lastname: 'Kooper'

};

}

In the above code snippet, we are changing the reference of the object instead of just mutating just one property.

Now when you run the application, you will find on the click of the button that the DOM is being updated with the new value.

By using **onPush Change Detection**, Angular will only check the tree if the reference passed to the component is **changed** instead of some **property changed** in the object. We can summarize that, and use an Immutable Object with **onPush Change Detection** to improve performance and run the change detector for the component tree when the object reference is changed.

We can further improve performance by using RxJS Observables because they emit new values without changing the reference of the object. We can subscribe to the observable for new value and then manually run ChangeDetector.

Let us modify AppComponent to pass an observable to MessageComponent.

import { Component, OnInit } from '@angular/core';

import { BehaviorSubject } from 'rxjs/BehaviorSubject';

@Component({

selector: 'app-root',

template: `

<app-message [person]='data'></app-message>

<button (click)='changeName()'>Change Name</button>

`

})

export class AppComponent implements OnInit {

p: any;

data: any;

ngOnInit(): void {

this.p = {

firstname: 'Brad',

lastname: 'Cooper'

};

this.data = new BehaviorSubject(this.p);

}

changeName() {

this.p = {

firstname: 'Foo',

lastname: 'Kooper'

};

this.data.next(this.p);

}

}

In the code, we are using BehaviorSubject to emit the next value as an observable to the MessageComponent. We have imported BehaviorSubject from RxJS and wrapped object p inside it to create an observable. On the click event of the button, it's fetching the next value in the observable stream and passing to MessageComponent.

In the MessageComponent, we have to subscribe to the person to read the data.

@Input() person: Observable<any>;

\_data;

ngOnInit(){

this.person.subscribe(data => {

this.\_data = data;

});

}

Now, on the click of the button, a new value is being created, however, a new reference is not being created as the object is an observable object.

Since a new reference is not created, due to onPush ChangeStrategy, Angular is not doing change detection.

In this scenario, to update the DOM with the new value of the observable, we have to manually call the change detector as shown below:

export class MessageComponent implements OnInit {

@Input() person: Observable<any>;

\_data;

constructor(private cd: ChangeDetectorRef) { }

ngOnInit() {

this.person.subscribe(data => {

this.\_data = data;

this.cd.markForCheck();

});

}

}

We have imported the ChangeDetectorRef service and injected it, and then called markForCheck() manually to cause change detector to run each time observable emits a new value.

Now when you run application, and click on the button, the observable will emit a new value and the change detector will update the DOM also, even though a new reference is not getting created.

To summarize:

* + 1. If Angular ChangeDetector is set to default then for any change in any model property, Angular will run change detection traversing the component tree to update the DOM.
    2. If Angular ChangeDetetor is set to onPush then Angular will run change detector only when new reference is being passed to the component.
    3. If observable is passed to the onPush change detector strategy enabled component then Angular ChangeDetctor has to be called manually to update the DOM.

In Angular, the trackBy feature is used to enhance the performance of rendering lists using ngFor. Let me explain how it works with an example.

* + By default, when you use ngFor to iterate over a collection of data, Angular will create and destroy DOM elements for each item in the collection.
  + However, with large datasets, this can lead to poor performance because even small changes (such as adding, editing, or deleting a record) cause Angular to recreate all associated DOM elements.
  + Here’s where trackBy comes in. It allows you to specify a unique identifier for each item in the collection.
  + [When the data changes, Angular can track which items have been added or removed based on this identifier and only update the relevant DOM elements1](https://codebriefly.com/how-to-use-trackby-in-angular-with-example/).

Let’s look at an example:

* + 1. First, create a data collection in your component (e.g., store) with items containing an id and a name.

import { Component } from '@angular/core';

@Component({

selector: 'my-app',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css'],

})

export class AppComponent {

name = 'Track By Example';

store: any;

constructor() {

this.getData();

}

getData() {

this.store = [

{ id: 1, name: 'John' },

{ id: 2, name: 'Doe' },

{ id: 3, name: 'Test' },

];

}

trackByFn(index: number, item: any) {

return item.id; // Use the item's ID as the unique identifier

}

getNewData() {

// Simulate adding new data

this.store = [

...this.store,

{ id: 4, name: 'Sam' },

{ id: 5, name: 'Kelly' },

{ id: 6, name: 'Thor' },

];

}

}

* 1. In your component’s HTML file (app.component.html), use trackBy in the ngFor loop:

<table>

<thead>

<tr>

<th>ID</th>

<th>Name</th>

<th>Action</th>

</tr>

</thead>

<tbody>

<tr \*ngFor="let item of store; trackBy: trackByFn">

<td>{{ item.id }}</td>

<td>{{ item.name }}</td>

<td><button (click)="deleteItem(item.id)">Delete</button></td>

</tr>

</tbody>

</table>

<button (click)="addItem()">Add item</button>

* 1. Implement the trackByFn method in your component to return the item’s id:

trackByFn(index: number, item: any) {

return item.id;

}

Now, when you add or remove items from the store collection, Angular will efficiently update only the necessary DOM elements based on their unique identifiers. [This significantly improves performance when dealing with large datasets1](https://codebriefly.com/how-to-use-trackby-in-angular-with-example/).